N87-26563

The Factorization of Large Composite Numbers on the MPP

by

Kathy J. McKurdy, Goodyear Aerospace Corporation
Marvin C. Wunderlich, Department of Defense

1. Introduction: The continued fraction method for factoring large integers (hereafter referred to as CFRAC) was an ideal algorithm to be implemented on a massively parallel computer such as the MPP. The history of this effort goes back many years. The first effort to implement this algorithm on the ILLIAC IV was thwarted first by an inadequate resolve on the part of a funding agency and then by the sudden dismantling of the computer itself. The second attempt was to put the program on the English DAP with the able assistence of Dennis Parkinson of Queen Mary College, London. This effort was spoiled by the inadequate amount of time the second author was able to spend in England in the summer of 1982. He was finally able to devote full time on the NASA MPP implementation in the summer of 1984 and by September of 1985, the authors suceeded to factor their first 60 digit number on the MPP using about 6½ hours of array time. Although this result added about 10 digits to the size number we could factor using CFRAC on a serial machine, it was already badly beaten by the implementation of Jim Davis and Diane Holdridge on the CRAY-1 using the quadratic sieve, an algorithm which is clearly superior to CFRAC for larger numbers. This work does illustrate, however, an algorithm which is ideally suited to the SIMD massively parallel architecture and we describe some of the modifications which were needed in order to make the parallel implementation effective and efficient.

2. The Continued Fraction Algorithm. To describe this method, we must first describe a method for generating small quadratic residues, mod N, where N is the composite number we wish to factor. An integer Q is said to be a quadratic residue, mod N, if an integer A exists such that

(1) $$Q \equiv A^2 (mod N).$$

Pairs (Q,A) satisfying (1) can be generated by expanding the simple continued fraction of $\sqrt{N}$. Space and time prevents us from elaborating on this subject so it must suffice to simply describe the algorithm. If we initiate the variables $d = [\sqrt{N}]$, $A_{-1} = 1$, $P_0 = 0$, $Q_0 = 1$, $A_0 = d$, we can generate the pair (Q_{k+1}, A_k) recursively from earlier pairs by the formulas

(2a) $$q_k = [(P_k + d)/Q_k]$$

(2b) $$P_{k+1} = q_k Q_k - P_k$$

(2c) $$Q_{k+1} = (N - P_{k+1}^2)/Q_k$$

and

(2d) $$A_{k+1} \equiv q_k A_k + A_{k-1} (mod N).$$

Then it can be proved that

(3) $$A_k^2 \equiv (-1)^{k+1} Q_{k+1} (mod N)$$

and

(4) $$Q_k \leq 2\sqrt{N}.$$

Now the clever reader may recognize a novel factoring method here. If N is the composite number to be factored, simply generate the pairs (Q_{k+1}, A_k) until Q_{k+1} is itself a square and k is even. Then if $Q_{k+1} = X^2$, we have $(A_k)^2 \equiv X^2$ (mod N) or

(5) $$N | A_k^2 - X^2 = (A_k - X)(A_k + X).$$

If N = pq where p and q are both primes, there is an even chance that one prime will divide $A_k - X$ and the other will divide $A_k + X$ and in this event, computing the greatest common divisor GCD(N,$A_k - X$) will reveal either p or q and N is factored. If this doesn't happen for Q_k, keep generating (Q_{k+1}, A_k) pairs until a square Q_{k+1} works. From (4), there are about $2\sqrt{N}$ different possible values of Q_k and among them there will be about $N^{.25}$ squares. Thus, we should have to generate about $N^{.25}$ values of Q_k before N is factored. The clever reader should congratulate himself for noticing this since his/her algorithm is already vastly superior to the simple divide and factor method which requires min(p,q) operations. However, this still consumes too much computer time to be really competitive with the leading state-of-the-art methods.

What we do in CFRAC is to obtain collections of Q's whose product is a square. Suppose I is a set of indices which defines such a collection. We deduce from (3) that

$$X^2 = \prod_{i \in I} (-1)^i Q_i \equiv \prod_{i \in I} A_{i-1}^2 = Y^2 (mod\, N) \quad (6)$$

and since $N = pq \mid X^2 - Y^2 = (X-Y)(X+Y)$, a factor can be produced by computing GCD(X − Y,N). To find a collection of Q's whose product is a square, we attempt to factor each Q over a fixed set of primes $p_1, p_2, \ldots, p_k$ and represent each Q_i which factors completely with a binary vector $(\varepsilon_0, \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k)$ where ε_0 is ± 1 according to which of $\pm Q_{i+1} \equiv A_i^2$ in (3) and ε_j, $i < 0$, is one if p_j divides Q_i to an odd power and zero otherwise. Note that the vector ε_i is the zero vector if and only if the corresponding value Q_i is a square and a quadratic residue, mod N. When we have factored more than k of these values Q, we can form a matrix M with these vectors and M will have more rows than columns. We can perform a Gaussian reduction on this matrix, produce zero rows and each such zero row will represent a collection of Q's whose product is a square. Thus (6) is satisfied and we may be able to factor N by computing the appropriate GCD. If this doesn't work, we can use another such collection of Q's since a collection whose product is a square is generated with each zero row produced from the Gaussian reduction.

3. The MPP Implementation: The most time consuming aspect of this algorithm is the factorization of the Q's. Each value of Q must be divided by each of the k primes in the base of primes until the number of factored Q's exceeds k. To factor a typical 60 digit number, one must attempt a factorization of over 100,000,000 values of Q using a base of 4,000 prime numbers and this requires 4×10^{11} division instructions. The MPP implementation described in this section performs this task very efficiently as well as the task of generating the (Q,A) pairs and the final Gaussian reduction. We shall discuss these three parts of the implementation separately.

3.1. Stepping. The generation of the (Q,A) pairs is clearly a recursive process and there is no obvious way to employ 16,384 processors to accomplish this task in parallel. However, Daniel Shanks and Hugh Williams have devised a very clever algorithm for taking giant steps in the recursion process. Knowing (Q_{s+1}, A_s) and (Q_{t+1}, A_t), one can generate a term (Q_{u+1}, A_u) whose u is very near s + t. The time needed for this composition is a constant which does not depend on s and t. This enables us to generate a pair (Q_{t+1}, A_t) where t is as large as we please by composing a succession of terms with a nearby term approximately $\log_2 t$ times. For this particular implementation, we first generated (Q_{t+1}, A_t) for t near one million (1M) and then generated 16,384 pairs (Q_{r+1}, A_r) where r was 1M, 2M, ..., 16384M. This was done on a fast serial machine. Then we put a pair (Q_{r+1}, A_r) in each of the 16,384 processors and generated successive terms in parallel using the recursion (2). Since the terms are 1M apart, we can generate as many as 16,384,000,000 terms before there is any danger of the same pair being generated in neighboring processors.

A serious problem arose in trying to implement the recursion described in (2) in parallel on the the MPP. The numbers involved are quite large. For a 60 digit factorization, the A's are 200 bits, the P's and Q's are each 100 bits and it was not possible to perform all the necessary arithmetic in the 900 bits of available memory in the ARU. For this reason, we used a fast bit plane I/O system developed by Goodyear Aerospace to use the staging memory as auxiliary storage. Using that package, storage can be allocated in the stager memory in the same way that storage is allocated in the ARU. A set of SEND macros exists which moves data between the stager and the ARU. SEND macros also exist to move data between the ARU and the Host, and the MCU and host. This package has essentially doubled the available memory for doing computational processing and has also provided an easy-to-use I/O management package for the entire algorithm. Data moves between the ARU and the stager can concur with computational operations which considerably reduces the extra time needed for the data swapping.

3.2. The Factoring. Having a different value of Q in each processor and the corresponding value A in the staging memory, the program now proceeds to attempt a factorization of the Q's over a set of primes stored as scalars in the MCU. Actually, the MCU only contains the differences between the consecutive primes. It also should be pointed out that the prime base consists of the smallest 4000 primes which are possible divisors of the Q's, and since the Q's are quadratic residues, mod N, only primes p for which N is a quadratic residue, mod p, are possible divisors of Q and so the prime base consists of the smallest 4000 primes having this property.

The fundamental operation is to divide all the values of Q by the integer p in one simultaneous ARU instruction and flag those P.E.'s where the remainder is zero. This operation is then repeated in the flagged processors, toggling a parity plane until all processors produce a non-zero remainder. The parity plane will contain a 1 or 0 indicating whether the primes p divided Q to an odd or even power. The difficulty with this method is that the divide instruction must be repeated t times where t is the largest power of p which divides any of the 16,384 values of Q. This can be rather large for small primes p. Certainly the divide instruction must be executed at least twice for each batch of Q's so the efficiency of the algorithm will be at most .5.

A two step algorithm is employed which avoides this difficulty. In the first step, the values Q are divided by each p in the prime base exactly once, and a table is collected in each PE which contains the set of primes p_{k_1}, p_{k_2}, ... , P_{k_l} which divides the Q in that particular processor. This table contains between 12 and 15 primes for each Q. Then the single step method described above is applied to the primes in the table in order to ascertain the parity of the power of p which exactly divides Q. This way the inefficiency of the single step procedure only affects about 15 division instructions rather than 4000. A serious difficulty arises, however, when attempting to implement this two step procedure. In the first step, we will be dividing 16,384 Q's by a scalar prime p setting a FLAG to 1 wherever the remainder is zero. Then in all processors in which FLAG = 1, the prime p must be put at the end of a short table in the ARU. However, the address of the end of the table is different in each processor. The "lock step" SIMD character of the MPP does not permit storing a value in different locations in different PE's.

We are indebted to Kenneth Batcher of Goodyear Aerospace for providing an ingenious solution to this problem. We begin by dividing the first 200 primes into the 16,384 Q's and setting 200 bit planes to flag the values Q which were evenly divisible by the primes. We now go back through the 200 bit planes and using them as flags, push onto the shift register the least significant 2 bits of the primes that evenly divide the Q's. At this point, the shift register in each PE contains the table p_{k_1}(mod 4), p_{k_2}(mod 4), ... p_{k_l}(mod 4) where the primes p_{k_i} are those which exactly divide the Q in that processor. The shift register is now stored in the least significant 2 bits of each entry of the table we are attempting to construct. Then the same procedure is followed for the 3rd and 4th significant bits, the 5th and 6th bits and so on until the 19th and 20th significant bits of the primes. No prime in the factor base ever exceeds 20 bits. The total number of bit instructions used in this complicated procedure is the same as if there were a variable address store in the SIMD instruction set. This routine was coded by K. Batcher in PEARL on the PECU. The value 200 was chosen because this number of bits was the most we could spare in the 1000 bit ARU. This procedure would be much easier to program on an MPP with larger memory.

This entire 3-step algorithm was executed by an MPP program called FACTOR and was able to do one complete batch of 16,384 values of Q in about one second of MPP time. Since the minimum number of division instructions needed to accomplish this task is 16,384 x 4000 = 65,536,000, this parallel routine operates at an average rate of 15.25 nanoseconds per instruction. The central instruction used in the program divided a 20 bit prime into a 100 bit Q and this used an optimally coded PEARL instruction which uses about two thousand 100 nanosecond cycles and this averages out to 12.21 nanoseconds per division instruction. From this, it follows that the FACTOR program operated at an efficiency rate of 12.21/15.25 or about 80%.

The program to generate the next batch of Q's and A's took considerably longer than was anticipated, mainly because of the tiny amount of core allocated to each P.E. Each STEP took about .1 seconds, but since it took so much less time than FACTOR, there was no driving need to optimize this procedure.

3.3. The Gaussian Reduction. There was no need to optimize this part of the program either . For the earlier factorizations, a VAX program was used to perform the reduction and it took about 25 minutes of VAX time. By way of comparison, over 6 hours of time was used to perform the factoring on the MPP. On the other hand, the very existence of 16,000 x 1,000 = 4,000 x 4000 bits of readily accessible memory made the development of an MPP-based Gaussian elimination program an irresistible temptation. For this purpose, it would have been preferable to have 4000 PE's each having 4000 bits of memory. Then, we could store each row of the matrix in one bit plane of the MPP. In this situation, we had to store one row of the matrix in a quarter of a bit plane and this was done by an arrangement of stripes in which columns 0, 4, 8, . . . , 124 of a 128 x 128 bit plane represented one row of the 0-1 matrix M. With this arrangement, an entire 4000 x 4000 matrix of bits can be stored in the entire MPP array memory, leaving very little memory for anything else.

The usual procedure to do a Gaussian reduction on a bit matrix is to do a series of elementary row operations until a matrix is obtained having a single 1-bit in each row and column -- i.e, a permutation of the identity matrix. One also performs the same elementary row operations on a history matrix which was set at the beginning to the identity matrix. If, at anytime in this process, a zero row is produced, the ones in the history matrix identify the rows on the original matrix which were initially linearly dependent. Of course, having the entire memory of the MPP used to store the original matrix M, there is no room for a history matrix. Therefore, we utilized an in-place algorithm first suggested to the author by Dennis Parkinson and completely described in [2]. We shall not give a detailed description of the procedure in this paper, but the idea is to use the "zero-space" produced by elementary row operations to store the "one-space" generated in the history matrix.

Theoretically, one should be able to always reduce a n by n bit matrix in $1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$ elementary row operations and if one row operation takes one MPP cycle of 100 nanoseconds and $n = 4000$, the time should be just under one second. Of course, the tight loop requires branch instructions and tests which themselves require at least 100 nanoseconds apiece. When the reduction program GELIM finally worked, a stop-watch timing of the program showed that usually 8 seconds were required to reduce the sparse matrix generated by CFRAC.

The operation of this program highlighted an interesting but disturbing feature of massively parallel processing. The factoring and Q generation portions of the program were completely fault tolerant. Every time a plane of (Q, A) pairs were generated, the relationship $\pm Q \equiv A^2 \pmod{N}$ was tested and those processors which failed the test were disabled with a mask for the rest of the run. On one occasion, there were 7 or 8 processors disabled after several hours of MPP computation. In GELIM, however, no such tolerance was permitted. If just one bit were at fault anywhere in the execution of the algorithm, the data obtained by GELIM was rendered completely useless. This actually occurred when the first factorization of a never-before-factored number was attempted. In August of 1985, The second author was scheduled to deliver a talk at a computational number theory conference in Arcata, California, and on the morning of the talk, GELIM was to triumphantly produce a set of linearly dependent rows of the matrix M. However, when the number of one bits in each column of the dependent set was counted, about 3% of the 4000 columns was odd, not even as required. Apparently, a single bit of the matrix was in error somewhere in the algorithm and by the end of the run, the fault spread to infect about 6% of the column data. He never gave that talk but rather rushed back to Goddard where he quickly put together a hasty GELIM on the VAX which produced the desired factors within 2 weeks of the aborted talk. It wasn't until last February when the first author found the hardware bug in the MCU which caused the occasional error and patched the MPP program so that it would run correctly each time.

Perhaps error correction is really needed in the ARU memory chip.

4. The Large Prime Variation. This is an improvement of the basic factoring strategy which has been utilized in all implementations of CFRAC. If, after a quadratic residue has been divided by all the admissible primes which are less than a number x, the remaining unfactored part F is less than x^2, then F itself must be a prime. Since F is not in the factor base, we call these large prime factorizations. If two different quadratic residues, Q_1 and Q_2, have large primes factorizations with the same large prime F, then the product $Q_1 * Q_2$ will have a factorization of the form

$$Q_1 Q_2 = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_j^{\alpha_j} F^2 .$$

where all the p_i are in the factor base. In the factoring process, this means that when Q_i and Q_j have the same large prime factorization and produce the 0-1 vectors ε_1 and ε_2, the exclusive OR of ε_1 and ε_2 can be added to the matrix M. In practice, very few pairs of large prime factorizations have the same large prime F when F is substantially larger than the largest prime in the factor base $p_k = x$. In the MPP implementation of this variation, all the large prime factorizations with $F < x^2/10$ were saved using a binary tree contained in the host VAX and whenever a collision occured between two Q's, the exclusive OR of the two variables was added to the matrix M. This variation was added to the MPP factoring program by the first author and despite all the additional overhead involved with the procedure, it improved the performance by nearly 1.5 for numbers in the 60 digit range. It is generally believed that the usefulness of this variation will diminish as the size of the number increases.

5. Results. The table below summarizes the results of five factorizations of four different large numbers. The first column defines the origin of the number which was factored. In every case, the expression in column 1 had some algebraic and small known factors which were divided out of the number before they were processed by the MPP. Column 2 indicates the size in decimal digits of the number after these smaller factors were divided out. Column 3 indicates whether or not a straight CFRAC was employed or the large prime variation, CFRAC-LP. Columns 4 and 5 indicate the computer time used. The large amount of VAX time needed for the first two factorizations was due to an inefficient proceedure for computing the product of the Q's. Columns 6 and 7 lists the total number of Q for which a factorization was attempted and the rate at which the Q's were processed. Note that the large prime variation reduced substantially the number of Q's needed for the total factorization.

References

1. M. A. Morrison and J. Brillhart. "A Method of Factoring and the factorization of F_7", *Math Comp.*,29(1975), 183 - 205

2. D. Parkinson and M. C. Wunderlich, "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", *Parallel Computing* , 1 (1984).

3. M. C. Wunderlich, "Implementing the continued fraction factoring algorithm on parallel machines", *Math. Comp.* 44(1985) 251-260.

FIVE FACTORIZATIONS ON THE MPP

Number	Digits	Method	MPP hours	Total hours	Q attempted	Qs/MPP sec
$2^{299}-1$	60	CFRAC	6.4	10.9	309,657,600	13,440
$2^{405}-1$	60	CFRAC	4.0	11.5	182,894,592	12,701
$5^{171}+1$	62	CFRAC	14.0	14.5	646,791,168	12,833
$2^{405}-1$	60	CFRAC-LP	2.8	3.4	93,028,352	9,120
$5^{149}+1$	64	CFRAC-LP	9.75	10.4	395,986,512	11,281